Add-and-Lambda II:
Eliminating Busy Waits

by

Malcolm C. Harrison

Ultracomputer Note #139
revised March 1988

Add-and-Lambda II: Eliminating Busy Waits

by

Malcolm C. Harrison

Ultracomputer Note #139
revised March 1988

This work was partially supported by IBM and DARPA under contract N00039-84-R-0605.

ABSTRACT

This paper describes a further extension of the combining fetch-and-add used in shared memory multiprocessors such as the NYU Ultracomputer and the IBM RPn. This extension permits the elimination of busy-waiting in a number of important parallel operations. These operations include barrier synchronization, stack and queue operations, full/empty bits, and group lock; most of these operations become one or two single-instruction operations for the PE. The hardware necessary is compatible with the design of the switches which are used in the Ultracomputer, and in fact are implemented as fetch-and-add instructions with an increment of unity.

1. Summary

In this paper we describe a relatively minor architectural addition to the combining fetch-and-add instruction which has been used on a number of shared memory machines. The main idea is to provide a (micro-) programmable interface between the network and the memory. This programmable interface (or memory controller), which we refer to as the add-and-lambda processor (or A&Lp) would receive from the network an address and an increment, and would be responsible for modifying the memory and returning the appropriate value to the network. The program executed by the A&Lp would normally be regarded as part of the system, and not normally accessible to users.

We show how this permits more efficient implementation of a number of parallel programming primitives, in some cases without busy-waiting. We give implementations of the following form:

barrier synchronization (few PEs)

```
F&A(addr,1);
```

queue insertion

```
q[F&A(addr1,1) mod n] := item;
F&A(addr2,1);
```

queue extraction

```
item := q[F&A(addr1,1) mod n];
F&A(addr2,1);
```

stack insertion

```
s[F&A(addr1,1)] := item;
F&A(addr2,1);
```

stack extraction

```
item := s[F&A(addr1,1)];
F&A(addr2,1);
```

read-when-full (1 reader)

```
v := F&A(addr1,0);
```

write-when-empty (I writer)

F&A(addr2,v);

None of these involve busy waiting. In addition, we show how code for the following operations:

lock

barrier synchronization (many PEs)

read-when-full / steal / lock (multiple readers)

can be implemented with code of the form:

```
if F&A(addr1,I) < > 0 then
   while F&A(addr2,I) < > 0 loop null endloop;
```

with less iterations of the busy-wait loop.

1.1. A Simple Example – Barrier Synchronization

Before going into details, we give an informal (and somewhat simplified) description of how the extensions we propose might work. We consider the barrier synchronization operation for a small number of PEs on a shared-memory multiprocessor with a combinable F&A operation. For this operation, each of a number n of processes must wait at a synchronization point in the program until the other n-1 reach such a point. Each PE does this by executing

F&A(a,1);

The effect of this instruction is to send a fetch-and-add request to memory location a. If a number (say m) of PEs execute this instruction at the same time, the combining fetch-and-add network will deliver to the memory a request which looks like:

F&A(a,m)

With the add-and-lambda approach, this request is processed by the add-and-lambda processor (A&Lp) in the following way. The A&Lp looks at location a to see what kind of an object it is. It finds that it is tagged as a barrier-synchronization object, with a count of n. If m is equal to n, it returns a value of zero through the network, so the n requesting PEs get distinct values for i between 0 and n-1. Since there were n requests, the PEs are synchronized, and can proceed.

If m is not equal to n, which is usually the case, the A&Lp does not respond to the request immediately, but puts the request in a queue of unanswered requests, and accumulates the number of such requests in a field (say c) in a. When the value of c reaches n, the A&Lp

responds to all the requests in the queue. In this case the values returned are not significant.

The result is a single-instruction barrier synchronization operation with no busy waiting. (Note that this solution is good for a small number of PEs, but not for a large number, since the responses to the requests will actually be issued serially by the A&Lp -- we discuss another solution for this problem below).

2. Introduction

The combining fetch-and-add (F&A) operation has been shown [GGKMRS83, GLR83, D85] to be capable of implementing a number of important parallel operations in a shared-memory multiprocessor. In simple operations, such as assigning the elements of a vector to different processors (PEs), where the assignments made by the combining network are correct, F&A gives good results. However, in more complicated cases, the solutions are sometimes inefficient and complex. This can be explained to some extent by the observation that the combining network does not have enough state information to make correct decisions, which may subsequently have to be corrected by the processors, requiring further coordination. Even in the relatively simple case of parallel queue operations, five accesses to the global memory are needed, and the algorithm contains highly non-intuitive code to avoid a tricky race condition.

In [H86] it was shown that many of these operations could be made considerably more efficient by moving some of the critical decision-making to a more logical place, namely to the network-memory interface. The operations required were more general than simply incrementing the value in the memory, and were called operations, or more simply A&L. In this paper we present an alternative formulation of these operations, which we will refer to as A&L2. The main advantage of this formulation is that it permits the possibility of eliminating busy-waits in some cases.

3. Add-and-Lambda

In the following sections, we describe three ways in which the A&Lp might be used.

3.1. Version 1

In A&L1, described in more detail in [H86], we considered a memory location to consist of a number of fields, and operations on these locations as F&As with the increment being a

value of 1 in one or more fields. These instructions will be combined by the network in the usual way, ending up at the memory location in the form of a positive increment. If the fields are large enough, this increment specifies unambiguously how many increments have been requested for each field. If the number passed back through the network is positive, the result is effectively a number of independent F&As. Furthermore, values passed back through the network in fields not used by the F&A operations will not be changed by the network, permitting the communication of state information back to the PEs.

Significantly more power can be provided by even a modest processor provided at the network-memory interface. This permits the operation on the memory location to be more complex than just a simple increment, and the value returned through the network to carry significantly more information back to the PEs. In [H86] examples are given of some primitives which can be reduced by a factor of 10 in complexity using this method, both in the number of network operations, and in the number of PE instructions.

The work in [H86] suggested that an A&L microcoded processor with less than 256 microwords and several relatively narrow adders would be adequate for many applications.

3.2. Version 2

Each A&L2 object resides in a single memory module, though it may occupy a number of memory locations. (Note that global memory is usually mapped so that successive addresses are in different memory modules, to reduce contention when PEs are processing linear structures. This would then require that the addresses used to store an A&L2 object not be adjacent, which could cause some inefficiency in storage management. In some machines, such as the IBM RP3, both scattered and sequential mapping schemes are available; however, it would be necessary for sequentially mapped addresses to be globally accessible).

Each operation on an A&L2 object is a F&A operation with three parameters, the address of the object, an opcode and an increment. F&A operations will be combined in the usual way by the PE-memory network, but only operations on the same object with the same opcode can be combined.

The function of the A&Lp can thus be thought of as a procedure which receives as its arguments the base address b, the opcode op, the increment from the network, and generates the value to be returned through the network. The A&Lp may also access and change the

contents of the memory module. Its overall action is as follows:

```
procedure A&L (b: in address; op: in opcode;
        inc: in nwword; rv: out nwword) is
    begin
        rv := lambda1(b,op,inc);
        b := lambda2(b,op,inc);
    end A&L;
```

where lambda1 and lambda2 are the functions we will be discussing.

On the Ultracomputer design, the standard combining network will only combine identical addresses, so we can get the effect we want by using spare address bits to specify the opcode. In this paper we will write such an operation in the form F&A(ObjectAddress+opcode,increment). All the examples we describe in this paper can be implemented by 16-byte objects, so there are 4 bits potentially useable for opcodes (though none of our examples has more than four opcodes).

The base word of each A&L object will be at the address obtained by masking out the opcode, and will be used to store the state of the object, which can be initialized by standard write operations. (Larger A&L objects could also be accomodated by ensuring that the base word contains state information which identifies the real base address of the object. Smaller objects could be represented by using the base word to specify how much, if any, of the address should be regarded as opcode).

3.3. Eliminating Busy-waiting

Many A&L-based algorithms require a PE to busy-wait in some conditions. This is clearly undesirable for two reasons: it wastes PE time, and it increases network traffic. We describe here how much of this busy-waiting can be eliminated.

In many of the algorithms we have looked at, the A&L processor is aware of which responses will require the PE to busy-wait. This busy-wait would not have occurred if requests (from other PEs) had arrrived previously. What we propose, therefore, is that the A&Lp be given the ability to postpone dealing with a request, by putting it on a queue of requests for the specified location. This queue can be stored in the local memory of the A&Lp, using standard linked list techniques. Each entry will contain the number of the requesting processor, and usually some indication of the appropriate response. Enough memory will be required in the worst case for one entry per processor, which will in general be negligible

compared with the size of memory handled by the A&Lp.

We discuss below the effect of the enqueing operation on the lengths of queues within the switches themselves.

4. An Example – the Two-way Lock

We illustrate the technique by giving as an example a primitive we call a two-way lock, called a readers-readers lock in [], and which is a variant of the group lock proposed by Dimitrovsky [D86].

4.1. Definition

The two-way lock provides four operations on an object b;

```
lock1 (b);
unlock1 (b);
lock2 (b);
unlock2 (b);
```

with the property that if a number of processes execute:

lock1 (b); code1; unlock1 (b);

or

lock2 (b); code2; unlock2 (b);

no instance of code1 will execute at the same time as an instance of code2, and there is no starvation. The two-way lock is thus similar to a readers-writers lock, but without writer preference, and without writer mutual exclusion.

4.2. Applications

The applications of two-way locks include all the applications of group locks described in [D86], including:

– parallel queues, with code1 being insert, and code2 being extract: it is straightforward to implement parallelizable versions of insert using F&A: the two-way lock keeps them separate from the extracts;

– parallel stacks, for the same reason;

– readers-writers, with no starvation: code1 is read, and code2 is write; serialization of the writers can be done by implementing lock2 to return the position of the process

within the subgroup; furthermore, redundant writes can be eliminated if only the processor who gets a 1 actually writes (see Appendix 1 for details).

5. Two-way Lock without Busy-waiting

For this lock we will need four operations, and the object will be in one of four states:

l1: lock1 accepting
u1: unlock1 accepting
l2: lock2 accepting
u2: unlock2 accepting

and if we refer to processes issuing locki requests as i-processes, will have the following structure:

```
type two_way_lock is
  record
  state: (l1,u1,l2,u2) := l1;
  current: integer := 0;              -- # i-processes currently accepted
  finished: integer := 0;             -- # i-processes which have unlocked
  other: integer := 0;          -- # non-i-processes waiting
  nextg: integer := 0;          -- # i-processes waiting
  otherq: queue := null;              -- queue of delayed i-processes
  nextgq: queue := null;              -- queue of delayed non-i-processes
  end;
```

The algorithm we use is as follows: lock1 requests are accepted when in state l1, returning the position in the current 1-group to the requesting PE. A lock2 request when in state l1 closes the 1-group, changing the state to u1; the response to this request is delayed until the current 1-group is finished. When in state u1, lock1 and lock2 requests are accumulated, but the responses are delayed until the appropriate groups are started. when the requesting PEs receive their position in the group. Also, in state u1, unlock1 requests are counted till they equal the number of accepted lock1 requests, at which time the state changes to l2, the queued lock2 requests can proceed, and the rejected lock1 requests become next in line. The algorithm is as follows (we use the notation a + = b for a := a+b):

```
case op of
   lock1:     service_lock1;
   unlock1:    service_unlock1;
   lock2:     service_lock2;          -- analogous to lock1
   unlock2:    service_unlock2;        -- analogous to unlock1
   end case;
```

where service_lock1 is the following:

```
        case state of
           l1: rv := current;  current + = inc;  reply(rv);
           u1: rv := nextg;  nextg + = inc;  enqueue(rv,nextgq);
           l2: if current = finished then
                  current := inc;  finished := 0;
                  state := l1;  reply(0);
               else
                  rv := other;  other + = inc;
                  state := u2;  enqueue(rv,otherq);
               endif;
           u2: rv := other;  other + = inc;
               enqueue(rv,otherq);
        end case;
```

and service_unlock1 is:

```
        case state of
           l1: rv := finished;  finished + = inc;  reply(rv);
           u1: rv := finished;  finished + = inc;  reply(rv);
               if current = finished then
                  current := other;  finished := 0;  replyall(otherq);
                  other := nextg;  otherq := nextgq;
                  nextg := 0;  nextgq := null;
                  if other = 0 then   {l2}
                     state := l2;
                  else               {u2}
                     state := u2;
                  endif;
               endif;
           l2, u2:
               error;
        end case;
```

The appropriate action for erroneous requests (unlocks without locks) will in general depend on the application, but could be to return a non-zero reject field.

The code for the lock2 operation is just F&A (b+lock2, 1) which returns the position in the group; the code for unlock2 is just F&A (b+unlock2, 1).

6. Two-way Lock with Busy-waiting

For technical reasons it is convenient to use 8 states, so a two-way-lock will have the following structure:

```
        type two_way_lock is
              record
              state: (0..7) := 0;
              current: integer := 0;
```

```
finished: integer := 0;
other: integer := 0;
nextg: integer := 0;
end;
```

In addition, when a request is rejected, we need to be able to use a field in the rv word to specify the state which the PE must wait for; we will use the notation rv.reject for this field. This field should be positioned so that as rv is passed back though the network, the value of rv.reject will not change, and rv will not go negative. It is convenient to make the rv.reject field 4 bits wide, so that values of the form (i+state), where i < 4, can be passed back. The PE can then use a non-zero test to detect rejection, and the lower 3 bits to specify the state to wait for. (Actually, the state specified for will be either an l1 or an l2, but the processor will also have to check for the corresponding u1 or u2, since the group may be closed immediately).

The algorithm we use is as follows: lock1 requests are accepted when in state l1, returning the position in the group to the requesting PE. A lock2 request when in state l1 closes the group, changing the state to u1; the response to this request specifies a non-zero state, whose lower 3 bits specify the state the PE must wait for. When in state u1, lock1 and lock2 requests are rejected, but their number is accumulated, and the requesting PEs receive the state they should wait for, and their position in the group. Also, in state u1, unlock1 requests are counted till they equal the number of accepted lock1 requests, at which time the state changes to l2, the rejected lock2 requests can proceed, and the rejected lock1 requests become next in line. The algorithm is as follows (we use the notation a += b for a := a+b):

```
case op of
   lock1:     service_lock1;
   unlock1:   service_unlock1;
   lock2:     service_lock2;          -- analogous to lock1
   unlock2:   service_unlock2;        -- analogous to unlock1
   end case;
```

where service_lock1 is the following:

```
case (state mod 4) of
   l1: rv := current;  current += inc;
   u1: rv := nextg;  nextg += inc;
      rv.reject := 3 + state;              {l1}
   l2: if current = finished then
         rv := 0;  current := inc;  finished := 0;
         state := (state + 2) mod 8;       {l1}
      else
```

```
            rv := other; other + = inc;
            rv.reject := state + 2;          {l1}
            state := (state + 1) mod 8;      {u2}
         endif;
      u2: rv := other; other + = inc;
         rv.reject := state + 2;             {l1}
   end case;
```

and service_unlock1 is:

```
   case (state mod 4) of
      l1: rv := finished; finished + = inc;
      u1: rv := finished; finished + = inc;
         if current = finished then
            current := other; finished := 0;
            other := nextg; nextg := 0;
            if other = 0 then   {l2}
               state := (state + 1) mod 4;
            else                {u2}
               state := (state + 2) mod 4;
            endif;
         endif;
      l2, u2:
         error;
   end case;
```

The code for the lock2 and unlock2 operations is more complicated than in the previous version, since it must be prepared for rejected requests; it can be written as follows:

```
   function lock2 (b: pointer to two_way_lock) returns integer is
         r : integer;
         w : integer;
      begin
      r := F&A (b+lock2, 1);
      if r.state < > 0 then              {busy wait}
         w := (r.state mod 8) / 2;       {ignore l.s. bit}
         r.state := 0;
         while w < > b.state/2 loop;
      end if;
      return r;
      end lock2;
   procedure unlock2 (b: pointer to two_way_lock) is
      begin
      F&A (b+unlock2, 1);
      end lock2;
```

This makes use of the state field in the returned value, with non-zero values used to indicate two of eight states (two iterations of the 4-state sequence) which the process must wait for.

7. Serialization

The non-busy-waiting implementation of the two-way-lock given above provides the operations in a single access to the global memory. The waiting time will be less than t(code1)+t(code2) in the best case, where t(codei) is the time to execute the code between locki and unlocki. However, there are two possible problems.

The first of these is that the queues in the combining switches could fill up (particularly those near the memory), limiting subsequent combinings. In general, this would suggest the use of longer queues, but there is no information available at the moment to suggest how long much longer, since the traffic pattern induced by the delayed response to F&A operations is dependent on algorithmic details.

The second is that the queues in the A&Lp could become long, resulting in the effective serialization of the responses. In the case of the two-way lock this will often not be a problem, since the lock is typically used relatively lightly. However, in the case of the barrier synchronization, a seemingly simpler problem, serialization of responses is more serious. A general solution, discussed in more detail in connection with barrier synchronization below, may be for the A&Lp to limit the length of the delayed-response queues. Responses would be queued till the queue reached a specific size, at which point the oldest request would be sent a rejection response. Busy-waiting would then be needed.

8. Hardware Multiprogramming in the PEs

A further improvement in performance is possible if the PEs are doing *hardware multiprogramming*, i.e. switching between processes while waiting for a response from the global memory. This technique has been suggested as a method for reducing the effect of network latency [e.g. SBK77, BS81]. In this case a delayed response to a synchronization request does not necessarily waste PE time.

9. Other Primitives

We give brief descriptions of implementations of a number of other parallel primitives.

9.1. Queue and Stack Operations

Parallel queue operations are important because their efficiency has an immediate influence on the granularity of processes. The structure described in [GLR83] uses a variant of

the usual circular array, with insertion and extraction pointers modified by F&A operations, and with upper bounds on the number of insertions and extractions in progress. In addition, a lock was required on each element.

A solution for this was given in [H87], which reduced the number of accesses, but which involved busy-waiting in several situations.

A non-busy-waiting implementation of queue operations can be obtained by use of the two-way-lock, as suggested above. However, it is also clear that there is some redundancy in this solution. We can remove this redundancy by storing the queue parameters (queue size, and insertion and extraction indices) in the A&L object. Since the lock operation in the two-way-lock algorithm returns the index within the current group, it can be modified to add the insertion (or extraction) index of the array used for the queue (or stack). This index would be updated when switching from inserting to extracting, and vice versa.

Queue overflow or underflow can be detected by the A&Lp, which can deal with it in a number of ways: by returning an illegal index; by not returning any index; or by abandoning the FIFO processing of requests. The best solution is perhaps a combination of the latter two: on overflow, close the current (inserting) group, and queue the request for the next inserting group; keep the next extracting group open till enough extractions have been accepted to permit a response to at least one of the delayed insertion requests.

The code to insert an item in a queue would then reduce to:

```
Queue[F&A(StartInsertAddress,1) mod QueueSize] := item;
F&A(InsertDoneAddress,1);
```

with similar code for extraction. This provides queue operations with no busy-waiting and no requirement for read-write interlocks.

Stack operations can be implemented similarly.

9.2. Readers/Writers

The standard F&A implementation of readers/writers uses PVchunk operations, P and V operations with a large value of increment. An alternative implementation using the two-way lock can be used instead.

```
type rwitem is  record
                twl: two_way_lock;
                contents: item;
```

```
            numwriters: integer := 0;
            end;
procedure read (a: rwitem) returns item is
      w: item;
   begin
      if lock1 (a.twl) = 0 then
         a.numwriters := 0;
      endif;
      w := a.contents;
      unlock1 (a.twl);
   end read;
procedure write (a: rwitem; w: item) is
      i: integer;
   begin
      i := lock2 (a.twl);
      while a.numwriters <> i loop;
      a.contents := w;
      a.numwriters + = 1;
      unlock2 (a.twl);
   end write;
```

A significant difference is that this solution would only require F&A operations with unit increments, since the opcode can be used to differentiate a read from a write.

9.3. Group Lock

The group lock, proposed by Dimitrovsky, is an object with three operations (glock, gsynch, and gunlock) which has the property that if a number of processes execute:

glock(g); A; gsynch(g); B; gunlock(g)

or

glock(g); A; gunlock(g);

no instance of A will execute at the same time as a B. Furthermore, there should be no serialization or starvation.

9.3.1. Implementation

The implementation of a group lock using A&L1 is described in [H86]; this is complicated by code to permit the PE to identify which state to busy-wait for. The implementation is considerably simplified if the A&Lp can queue responses, since no busy-waiting is required at all. We can represent the state of a group lock by an object which contains the state of the lock

(open, closed, synched), the number of PEs still in the current group, the number of PEs in the current group which have executed a synch operation, and a queue of waiting requests to enter the group. and the number of PEs which have been assigned to the next group.

For a group lock at base address b, a glock request will be implemented as F&A(b+glock,1), gsynch as F&A(b+gsynch,1), and gunlock as F&A(b+gunlock,1). The PE does not have to check the response, since all requests are accepted; a lock request is accepted either into the current group or the next, but in the latter case the response is only sent when it is in the group. Acceptance into the current group is permitted if no synchs or unlocks have been requested. The group lock object has the following components:

```
type glocktype is
      record
      state: (open, closed, synch) := open;
      current: integer := 0;
      synched: integer := 0;
      nextg: queue := nil;
      end;
```

The algorithm is as follows (we use the notation a += b for a = a+b and (x?y:z) for if x then y else z):

```
case op of
   glock:
      case state of
         open:              current += inc; reply(ok);
         closed, synch:      enqueue(nextg);
      end case;
   gsynch:
      synched += inc;
      state := (synched=current ? synch: closed);
      reply(ok);
   gunlock:
      current -= inc;
      if current = 0 then
         replyall(nextg); nextg := nil;
         current := 0; synched := 0; state := open;
      elsif synched = current then
         state := synch;
      end if;
   end case;
```

Here there are three active states, open, closed, and synch. The state is open if no gsynch requests have been received, closed if insufficient gsynch requests have been received, and

synch if insufficient gunlocks. (Note that this implementation differs from that in [D86], in that a group is only closed if a synch request is received; this eliminates PEs having to wait unnecessarily). This code does not return position within the group, though this could easily be added.

9.4. Single-PE Interlocking Reads and Writes (Full/Empty Flags)

A common synchronization primitive, often implemented in the hardware (e.g. the HEP's wait-and-read, the Butterfly's/Monarch's steal operation) associates a full/empty flag with a memory word, and provides operations which permit this to be read only if full, or written only if empty.

If used for communication between no more than two processors, these are simply provided by F&A operations with different opcodes. The F&A object would contain a data field and a full/empty bit. A read will be executed as a F&A(ReadAddress,0), and a write as F&A(WriteAddress,NewValue). These will not combine, and the F&A processor can hold up the response till the appropriate condition holds.

9.5. Lock and Unlock

A not-very-busy-waiting lock and unlock can be implemented as follows. The A&Lp would maintain two queues, a waiting-queue of requests that arrived when the lock was locked, and an accepted-queue of requests that were accepted but not yet satisfied. The lock operation would be F&A(LockAddress+lock,1), which could be combined. If the lock was locked, the A&Lp would put the request on the waiting-queue. If unlocked, the A&Lp would return a zero, and save the number of requests; the PE receiving a zero response would have the lock, and PEs receiving a non-zero value would send a F&A(LockAddress+retry,1), which would be combined and queued (on the accepted-queue) if the lock was still locked, and processed like a lock request otherwise (but before the requests on the waiting-queue). The lock code would be of the form:

```
if F&A(LockAddr+lock,1) < > 0 then
        while F&A(LockAddr+retry,1) < > 0 loop null endloop;
```

Here retries are only used in the case of requests which combine.

The unlock operation would be F&A(LockAddress+unlock,0) which would permit the A&Lp to respond to a delayed request on the accepted-queue. If the accepted-queue was

empty and the correct number of retries had been accepted, the A&Lp would process the waiting-queue. Processing the correct number of retries ensures fairness, since requests are processed in FIFO order.

Note that serialization is not a problem here, since this operation is inherently serializing (and should be used only as a last resort anyway).

9.6. Multiple-PE Read-when-full / Steal

This operation is like lock, but returns a value to one requesting PE; this value is returned in another field. The PE code could be written:

```
v := F&A(DataAddr+steal,1);
if (v & CountMask) < > 0 then
      repeat
            v := F&A(DataAddr+retry,1);
            until (v & CountMask) = 0;
```

The write operation resets the full/empty bit, and is just:

```
F&A(DataAddr+writefill,value);
```

The assumption is that only one PE (that having stolen the value) will be writing, so there is no problem with combining.

Note that the implementation of a multiple-PE write-when-empty operation is more difficult, because the data being written can not in general be combined, and there is no simple way to prevent it. If multiple writes are necessary, a separate lock will have to be used.

9.7. Barrier Synchronization

This problem was chosen as a simple example of how A&L can avoid busy-waits. However, the naive solution presented above may encounter problems if there are a large number of PEs synchronizing, since the response queue effectively serializes the responses by the A&Lp. A better solution is for the A&Lp to limit the size of the delayed-response queue by responding negatively to an old request when a new one is queued. The A&Lp could return a value BigNum (larger than the number of PEs) to indicate non-synchronization. The PEs would then submit a retry request, so the code would be:

```
if F&A(BarrierAddr+synch,1) > = BigNum then
      while F&A(BarrierAddr+retry,1) > BigNum loop null endloop;
```

The A&Lp will keep a count of retry requests as well as synch requests, and not start

processing any synch requests while there are PEs whose retry requests have not been accepted (the synch requests might be for the next barrier synchronization).

This version of barrier synchronization does busy-wait, but the busy-wait cycle is longer. Although it does not put a clear bound on the amount of serialization caused by the A&Lp queue, its performance would seem to be worthy of study.

10. Simplification of the combining F&A switch

We note that most of the above operations are implemented by a F&A operation with an increment of 1, so it is possible that a simpler implementation of the combining switch may be possible than in the standard F&A algorithms. For example, the standard F&A implementation of readers/writers need an increment of N (the number of PEs); with A&L the write request can be communicated by using a different address.

11. Problems

11.1. Interrupts

One defficiency of implementing operations without busy-waiting is that the PE may not be able to respond to interrupts while waiting. If the expected waiting time is short, this should create no more problems than masking interrupts in single-processor systems. Furthermore, many operations might reasonably be required to be indivisible; for example, context switching in the middle of a queue operation might block all use of the queue, and in the middle of a barrier synchronization operation might cause other PEs to wait.

In some cases it may be necessary to make use of a busy-waiting form of the operation to permit interrupts. One way of doing this might be to require that the A&Lp respond within a certain specified time. This could be implemented in a number of ways: a PE could send time-out signals in the form of A&L operations to the A&Lps; the A&Lp could maintain and interrogate a counter (corresponding to an approximate time); or the A&Lp could be provided with a timer and interrupt mechanism.

11.2. Errors

The ability of the A&Lp to delay responses to F&A operations introduces a new class of programming errors. These are errors which result in PEs which are deadlocked waiting for responses from the A&Lp which are never sent; For example, if a PE does a start-queue-insert

but does not follow it with the corresponding end-queue-insert operation, all PEs waiting for access to the queue will be deadlocked. To handle this problem, we require a mechanism to detect it, and a mechanism to recover from it.

Detection can be provided by an A&L object of type 'status', which will have one operation on it: F&A(StatusAddr+Inquire,PE) will return the status of the specified PE. More specifically, it will return the address and operation that the PE is expecting a response to, and zero otherwise. This operation will not be combinable, for obvious reasons, so its use must be protected by a lock. There must be at least one status object for every A&Lp.

Recovery is a little trickier, because the code in the PE has (presumably) been written to assume that there is no busy-waiting, and that any response will have satisfied the original request. It is straightforward to implement an A&L object to accept an operation of the form F&A(CommandAddr+Release,PE), which will force the A&Lp to return an error response to a PE with a delayed request. The problem is that this response may cause the PE to make further, and possibly disasterous errors.

There are two obvious solutions: to provide an error response to every A&L operation, which the PE can test for (at the expense of some inefficiency, perhaps); or by providing a hardware error signal which will cause a trap in the PE.

12. Acknowledgements

The author's attention was drawn to the busy-wait problem by Greg Pfister. Eric Freudenthal pointed out the problem of serializing responses in the barrier synchronization example.

13. References

[BS81]

Burton Smith, "Architecture and Applications of the HEP Multiprocessor Computer System", in Real-time Signal Processing IV, SPIE, Bellingham, Washington, 1981.

[D85]

Isaac Dimitrovsky, "Parallel Garbage Collection", Ultracomputer Software Note ??, Courant Institute, 1985.

[D86]

Isaac Dimitrovsky, "A Group Lock Algorithm, with Applications", Ultracomputer Note ??, Courant Institute, March 1986.

[GGKMRS83]

A Gottlieb, R Grishman, C P Kruskal,

K P McAuliffe, L Rudolph, M Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Computer", IEEE Trans. on Computers, February 1983.

[GLR83]

A Gottlieb, B D Lubachevsky, L Rudolph, "Coordinating Large Numbers of Processors", TOPLAS, January 1983.

[H86]

M C Harrison, "The Add-and-Lambda Operation: an Extension of Fetch-and-Add", NYU Ultracomputer Note 104, June 1986.

[H87]

M C Harrison, "More Uses of Add-and-Lambda", NYU Ultracomputer Note, January 1987 (in preparation).

[SPK77]

H Sullivan, T Bashkow, D Klappholz, "A Large Scale Homogeneous, Fully Distributed Parallel Machine", Proc 4th Ann Symp on Comp Arch, 1977.